01｜领域驱动设计到底在讲什么？

2021-06-23 徐昊

《如何落地业务建模》 课程介绍 >

讲述：徐昊
时长 20:23　大小 18.68M

你好，我是徐昊。今天我们来聊聊领域驱动设计（Domain Driven Design，即 DDD）。

说起业务建模，领域驱动设计是一个绕不过去的话题。自从 Eric Evans 在千禧年后发布他的名著“Domain Driven Design：Tackling the Complexity in the Heart of Software”，领域驱动设计这一理念迅速被行业采纳，时至今日仍是绝大多数人进行业务建模的首要方法。

有意思的是，可能因为成书年代过于久远，大多数人并没有读过 Eric 的书，而是凭直觉本能地接受了领域驱动这一说法，或是在实践中跟随周围的实践者学习使用它。但是**对于 Eric 到底在倡导一种什么样的做法并不了然。**

所以今天这节课，我们要回顾一下领域驱动设计的要点和大致做法，从而可以更好地理解 DDD 从何处而来，以及 DDD 在其创始人的构想中是如何操作的。

领域模型对于业务系统是更好的选择

我们都知道，软件开发的核心难度在于处理隐藏在业务知识中的复杂度，那么模型就是对这种复杂度的简化与精炼。所以从某种意义上说，Eric 倡导的领域驱动设计是**一种模型驱动的设计方法**：**通过领域模型**（Domain Model）**捕捉领域知识，使用领域模型构造更易维护的软件。**

模型在领域驱动设计中，其实主要有三个用途：

1. 通过模型反映软件实现（Implementation）的结构；
2. 以模型为基础形成团队的统一语言（Ubiquitous Language）；
3. 把模型作为精粹的知识，以用于传递。

这样做的好处是显而易见的：

1. 理解了模型，你就会大致理解代码的结构；
2. 在讨论需求的时候，研发人员可以很容易明白需要改动的代码，并对风险与进度有更好的评估；
3. 模型比代码更简洁，毕竟模型是抽象出来的，因而有更低的传递成本。

模型驱动本身并不是什么新方法，像被所有人都视为编程基本功的数据结构，其实也是一系列的模型。我们都知道有一个著名的公式“程序 = 算法 + 数据结构”，实际上这也是一种模型驱动的思路，指的是从数据结构出发构造模型以描述问题，再通过算法解决问题。

在软件行业发展的早期，堆、栈、链表、树、图等与领域无关的模型，确实帮我们解决了从编译器、内存管理到数据库索引等大量的基础问题。因此，无数的成功案例让从业人员形成了一种习惯：**将问题转化为与具体领域无关的数据结构，即构造与具体领域无关的模型。**

而领域驱动则是对这种习惯的挑战，它实际讲的是：**对于业务软件而言，从业务出发去构造与业务强相关的模型，是一种更好的选择。**那么模型是从业务出发还是与领域无关，关键差异体现在人，而不是机器对模型的使用上。

构造操作系统、数据库等基础软件的团队，通常都有深厚的开发背景，对于他们而言，数据结构是一种常识。更重要的是，这种常识并不仅仅体现在对数据结构本身的理解上（如果仅仅是结构那还不能算难以理解），还体现在与数据结构配合的算法，这些算法产生的行为，以及这些行为能解决什么问题。

比如树（Tree）这种非常有用的数据结构，它可以配合深度优先（Depth-First）、广度优先（Breadth-First）遍历，产生不同的行为模式。那么当开发人员谈论树的时候，它们不仅仅指代这种数据结构，还暗指了背后可能存在的算法与行为模式，以及这种行为与我们当前要解决的业务功能上存在什么样的关联。

但是，如果我们构造的是业务系统，那么团队中就会引入并不具有开发背景的业务方参与。那么这个时候，与领域无关的数据结构及其关联算法，由于业务方并不了解，那么在他们的头脑中也就无法直观地映射为业务的流程和功能。这种认知上的差异，会造成团队沟通的困难，从而破坏统一语言的形成，加剧知识传递的难度。

于是在业务系统中，**构造一种专用的模型（领域模型），将相关的业务流程与功能转化成模型的行为，就能避免开发人员与业务方的认知差异。**这也是为什么我们讲，领域模型对于业务系统是一种更好的选择。

或许在今天看起来，这种选择是天经地义的。但事实是，**这一理念的转变开始于面向对象技术的出现，而最终的完成，则是以行业对 DDD 的采纳作为标志的。**

不同于软件行业对数据结构的长时间研究与积累，在不同的领域中该使用什么样的领域模型，其实并没有一个现成的做法。因而**在 DDD 中，Eric Evans 提倡了一种叫做知识消化**（Knowledge Crunching）**的方法帮助我们去提炼领域模型。**这么多年过去了，也产生了很多新的提炼领域模型的方法，但它们**在宏观上仍然遵从知识消化的步骤。**

知识消化的五个步骤

知识消化法具体来说有五个步骤，分别是：

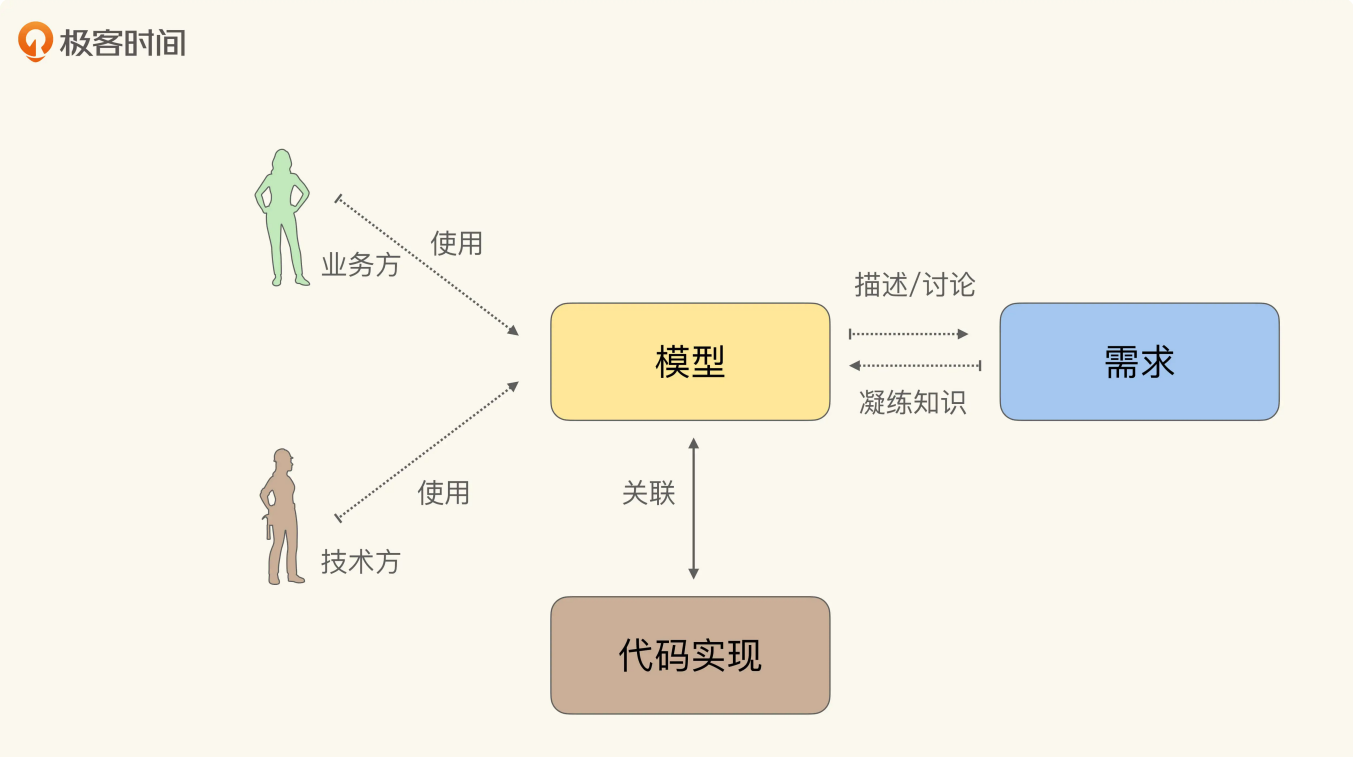
1. 关联模型与软件实现；
2. 基于模型提取统一语言；
3. 开发富含知识的模型；
4. 精炼模型；
5. 头脑风暴与试验。

在知识消化的五步中，**关联模型与软件实现，是知识消化可以顺利进行的前提与基础**。它将模型与代码统一在一起，使得对模型的修改，就等同于对代码的修改。

而**根据模型提取统一语言**，则会将业务方变成模型的使用者。那么通过统一语言进行需求讨论，实际就是通过模型对需求进行讨论。

后面三步呢，构成了一个**提炼知识的循环**：通过统一语言讨论需求；发现模型中的缺失或者不恰当的概念，精炼模型以反映业务的实践情况；对模型的修改引发了统一语言的改变，再以试验和头脑风暴的态度，使用新的语言以验证模型的准确。

如此循环往复，不断完善模型与统一语言。因其整体流程与重构（Refactoring）类似，也有人称之为**重构循环**。示意图如下：



说句题外话，目前很多人把 Knowledge Crunching 翻译为“知识消化”。不过在我看来，应该直译为“知识吧唧嘴”更好些，Crunching 就是吃薯片时发出的那种难以忽略的咔嚓咔嚓声。

你看，Knowledge Crunching 是一个如此有画面感的词汇，这就意味着当我们获取领域知识的时候，要大声地、引人注意地去获得反馈，哪怕这个反馈是负面的。

而且如果我们把它叫做“知识吧唧嘴”，我们很容易就能从宏观上理解 Knowledge Crunching 的含义了：吸收知识、接听反馈——正如你吃薯片时在吧唧嘴一样。

好了，言归正传，通过以上的分析，我们其实可以把“知识消化”这五步总结为**“两关联一循环”**：

“两关联”即：模型与软件实现关联；统一语言与模型关联；

“一循环”即：提炼知识的循环。

今天我们先介绍模型与软件实现关联。后面两节课，再关注统一语言与提炼知识的循环。

模型与软件实现关联

我们已经知道，领域驱动设计是一种模型驱动的设计方法。那么很自然地，我们可以得到这样一个结论：

模型的好坏直接影响了软件的实现；

模型的好坏直接影响了统一语言；

模型的好坏直接影响了传递效率与成本。

但 Eric Evans 在知识消化中并没有**强调模型的好坏，反而非常强调模型与软件实现间的关联**，这是一种极度违反直觉的说法。

这种反直觉的选择，背后的原因有两个：一是知识消化所倡导的方法，它本质上是一种**迭代改进的试错法**；第二则是一些**历史原因**。

所谓迭代改进试错法，就是不求一步到位，但求一次比一次好。正如我们刚才总结的，知识消化是“两关联一循环”。通过提炼知识的循环，技术方与业务方在不断地交流与反馈中，逐步完成对模型的淬炼。

无论起点多么低，只要能够持续改进，总有一天会有好结果的。而能够支撑持续改进基础的，则是实现方式与模型方式的一致。所以**比起模型的好坏**（总是会改好的），**关联模型与软件实现就变得更为重要了**。

历史原因则有两点：一是因为在当时，领域模型通常被认为是一种分析模型（Analysis Model），用以定义问题的，而无需与实现相关。这样做的坏处呢，我们下面再细讲。

二是因为当时处在面向对象技术大规模普及的前夕，由于行业对面向对象技术的应用不够成熟，将模型与实现关联需要付出额外的巨大成本，因而通常会选择一个相对容易、但与模型无关联的实现方式。这个相对容易的方式，往往是过程式的编程风格。

而与模型关联的实现方法，也就是被称作“富含知识的模型（Knowledge Rich Model）”，是一种面向对象的编程风格。因此，我们强调模型与实现关联，实际上也就在变相强调**面向对象技术在表达领域模型上的优势**。接下来我们具体分析。

从贫血模型到富含知识的模型

在 DDD 出版的年代，Hibernate（一种 Object Relationship Mapping 框架，可以将对象模型与其存储模型映射，从而以对象的角度去操作存储）还是个新鲜事物。大量的业务逻辑实际存在于数据访问对象中，或者干脆还在存储过程（Store Procedure）里。

如果把时光倒回到 2003 年前后，程序的“常规”写法和 DDD 提倡的关联模型与实现的写法，在逻辑组织上还是有显而易见的差异的。

我们现在考虑一个简单的例子，比如极客时间的用户订阅专栏。我们很容易在头脑中建立起它的模型：

在 ORM 流行起来之前的 2003 年（当然那时候没有 try-close 语法），如下的代码并不是不可接受：

复制代码

```
class UserDAO {

  ...
  public User find(long id) {
    try(PreparedStatement query = connection.createStatement(...)) {
      ResultSet result = query.executeQuery(....);
      if (rs.next)
         return new User(rs.getLong(1), rs.getString(2), ....);
      ....
    } catch(SQLException e) {
       ...
    }
  }
}

class SubscriptionDAO {
  ...
  // 根据用户Id寻找其所订阅的专栏
  public List<Subscription> findSubscriptionsByUserId(long userId) {
     ...
  }

  // 根据用户Id，计算其所订阅的专栏的总价
  public double calculateTotalSubscriptionFee(long userId) {
     ...
  }
}
```

这样的实现方式就是被我司首席科学家 Martin Fowler 称作**“贫血对象模型”**（Anemic Model）的实现风格，即：**对象仅仅对简单的数据进行封装，而关联关系和业务计算都散落在对象的范围之内。**这种方式实际上是在沿用过程式的风格组织逻辑，而没有发挥面向对象技术的优势。

与之相对的则是“**充血模型**”，也就是**与某个概念相关的主要行为与逻辑，都被封装到了对应的领域对象中。**“充血模型”也就是 DDD 中强调的**“富含知识的模型"**。不过作为经历那个时代的程序员，以及 Martin Fowler 的同事来说，“充血模型”是我更习惯的一个叫法。

Eric 在 DDD 中总结了构造“富含知识的模型”的一些关键元素：实体（Entity）与值对象（Value Object）对照、通过聚合（Aggregation）关系管理生命周期等等。按照 DDD 的建议，刚才那段代码可以被改写为：

复制代码

```
class User {

    // 获取用户订阅的所有专栏
    public List<Subscription> getSubscription() {
       ...
    }

    // 计算所订阅的专栏的总价
    public double getTotalSubscriptionFee() {
       ...
    }
}

class UserRepository {
   ...
   public User findById(long id) {
      ...
   }
}
```

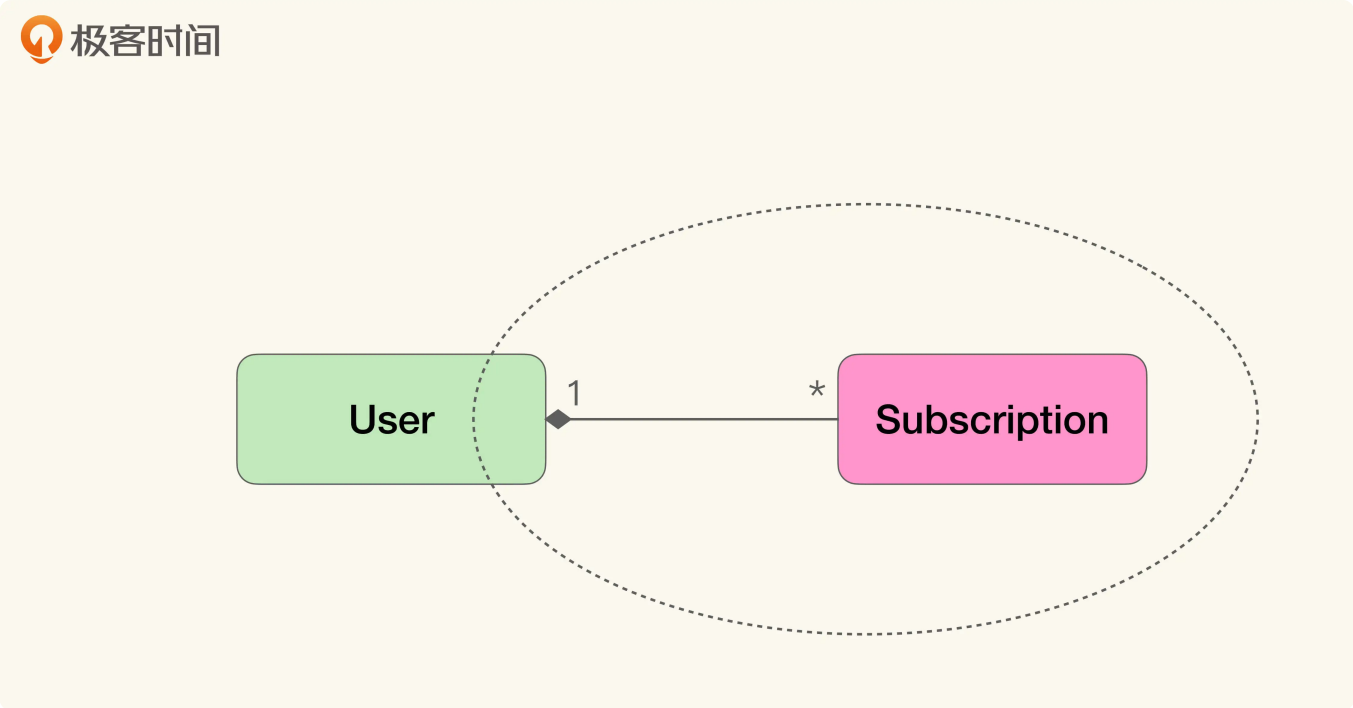
从这段代码很容易就可以看出：User（用户）是聚合根（Aggregation Root）；Subscription（订阅）是无法独立于用户存在的，而是被聚合到用户对象中。

通过聚合关系表达业务概念

不同于第一段代码中单纯的数据封装，改写后这段代码里的 User，具有更多的逻辑。Subscription 的生命周期被 User 管理，无法脱离 User 的上下文独立存在，我们也无法构造一个没有 User 的 Subscription 对象。

而在之前的代码示例中，我们其实可以很容易地脱离 User，直接从数据库中查询出 Subscription 对象（通过调用 findSubscriptionsByUserId）。所有与 Subscription 相关的计算，其实也被封装在 User 上下文中。

这样做有什么好处呢？首先我们需要明白，**在建模中，聚合关系代表了什么含义**，然后才能看出“贫血模型”与“富含知识的模型”的差异。我们还是以极客时间的专栏为例。



为了表示用户订阅了某个专栏，我们需要同时使用“用户”与“订阅”两个概念。因为一旦脱离了“订阅”，“用户”只能单纯地表示用户个人的信息；而脱离了“用户”，“订阅”也只能表示专栏信息。那么只有两个放在一起，才能表达我们需要的含义：用户订阅的专栏。

也就是说，在我们的概念里，与业务概念对应的不仅仅是单个对象。**通过关联关系连接的一组对象，也可以表示业务概念，而一部分业务逻辑也只对这样的一组对象起效。**但是在所有的关联关系中，聚合是最重要的一类。**它表明了通过聚合关系连接在一起的对象，从概念上讲是一个整体。**

以此来看，当我们在这个例子里，谈到 User 是 Subscription 的聚合根时，实际上我们想说的是，在表达“用户订阅的专栏”时，User 与 Subscription 是一个整体。如果将它们拆分，则无法表示这个概念了。同样，计算订阅专栏的总价，也只是适用于这个整体的逻辑，而不是 Subscription 或 User 独有的逻辑。

总结来说，我们无法构造一个没有 User 的 Subscription 对象，也就是说这种概念在软件实现上的映射，比起“贫血模型”的实现方式，“富含知识的模型”将我们头脑中的模型与软件实现完全对应在一起了——无论是结构还是行为。

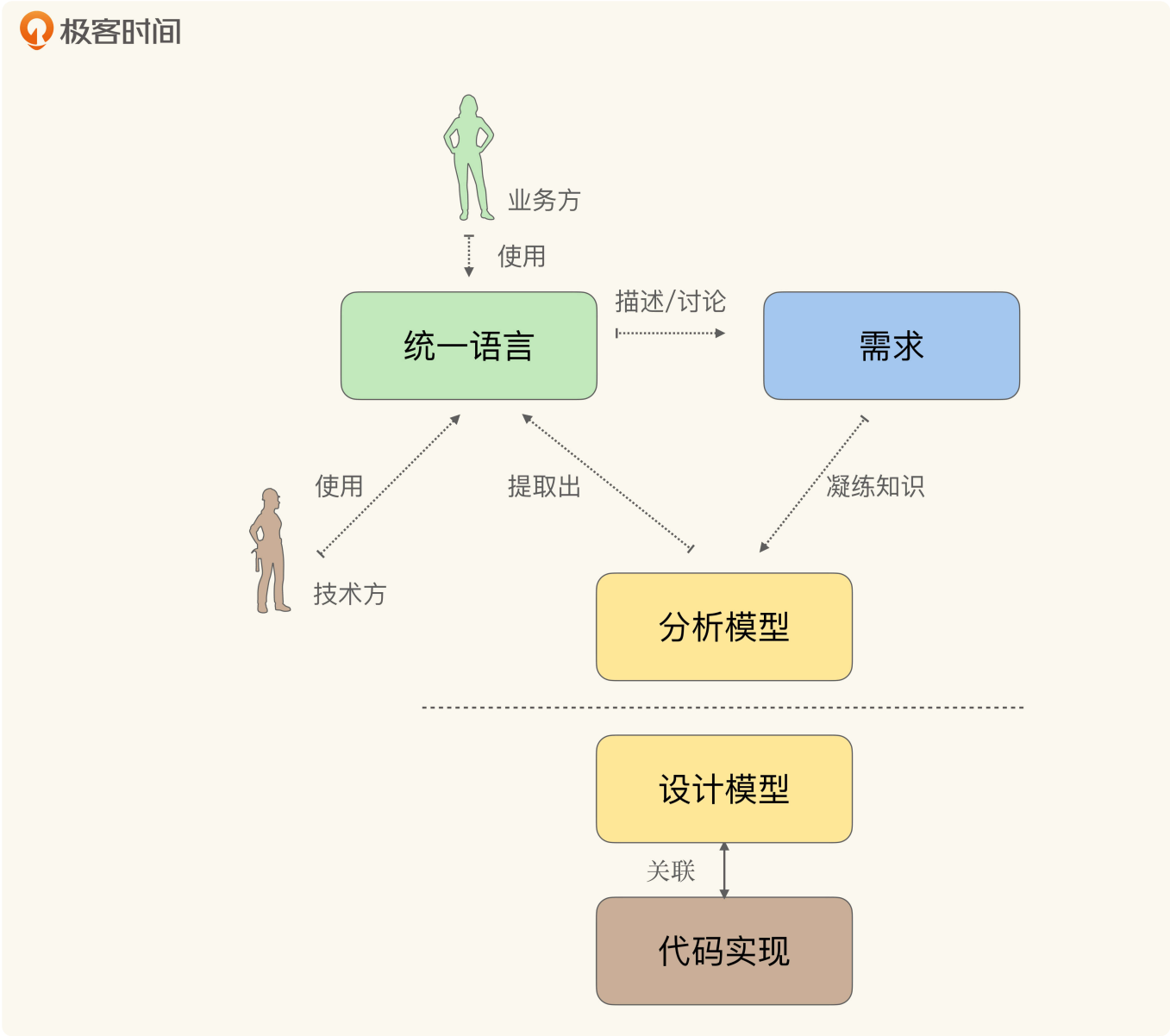
这显然简化了理解代码的难度。只要我们在概念上理解了模型，就会大致理解代码的实现方法与结构。同样，也简化了我们实现业务逻辑的难度。通过模型可以解说的业务逻辑，大致也知道如何使用“富含知识的模型”在代码中实现它。

修改模型就是修改代码

关联模型与软件实现，最终的目的是为了达到这样一种状态：**修改模型就是修改代码；修改代码就是修改模型**。

在知识消化中，提炼知识的重构是围绕模型展开的。如果对于模型的修改，无法直接映射到软件的实现上（比如采用贫血模型），那么凝练知识的重构循环就必须停下来，等待这个同步的过程。

如果不停下来等待，模型与软件间的割裂，就会将模型本身分裂为更接近业务的分析模型，以及更接近实现的设计模型（Design Model）。这个时候，分析模型就会逐渐退化成纯粹的沟通需求的工具，而一旦脱离了实现的约束，分析模型会变得天马行空，不着边际。如下所示，分析模型参与需求，设计模型关联实现：



事实上，这套做法在上世纪 90 年代就被无数案例证明难以成功，于是才在 21 世纪初有了模型驱动架构（Model-Driven Architecture）、领域驱动设计等一系列使用统一模型的方法。那么，在模型割裂的情况下，统一语言与提炼知识循环也就不会发生了，所以我们才必须将**模型与软件实现关联**在一起，这也是为什么我们称它是知识消化的基础与前提。

你或许会有疑惑，“富含知识的模型”的代码貌似就是我们平常写的代码啊！是的，随着不同模式的 ORM 在 21 世纪初期相继成熟，以及面向对象技术大规模普及，将领域模型与软件实现关联，在技术上已经没有多大难度了。虽然**寻找恰当的聚合边界**仍然是充满挑战的一件事，但总体而言，我们对这样的实现方式并不陌生了。

由此我们可以更好地理解，DDD 并不是一种编码技术，或是一种特定的编码风格。有很多人曾这样问我：**怎么才能写得 DDD 一点**？

我一般都会告诉他，**只要模型与软件实现关联了，就够了。**

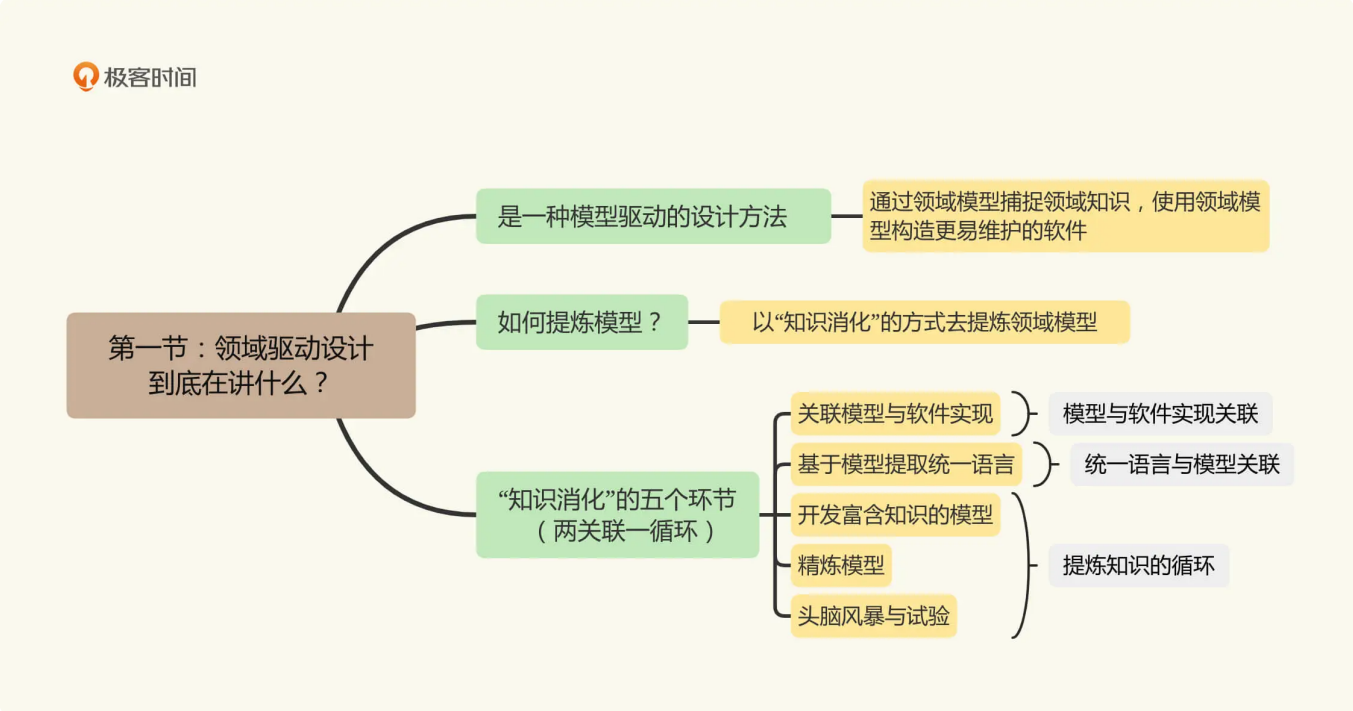
毕竟“DDD 的编码”的主要目的是不影响反馈的效率，保证凝练知识的重构循环可以高效地进行。如果不配合统一语言与提炼知识循环，那么它就只是诸多编码风格之一，难言好坏。

而如果想“更加 DDD”的话，则应该**更加关注统一语言与提炼知识循环**，特别是提炼知识循环。事实上，它才是 DDD 的核心过程，也是 DDD 真正发挥作用的地方。

小结

领域驱动设计是一种领域模型驱动的设计方法，它强调了在业务系统中应该使用与问题领域相关的模型，而不是用通用的数据结构去描述问题。这一点已被行业广泛采纳。

Eric Evans 提倡的知识消化，总结起来是“两关联一循环”：模型与软件实现关联；统一语言与模型关联；提炼知识的循环。



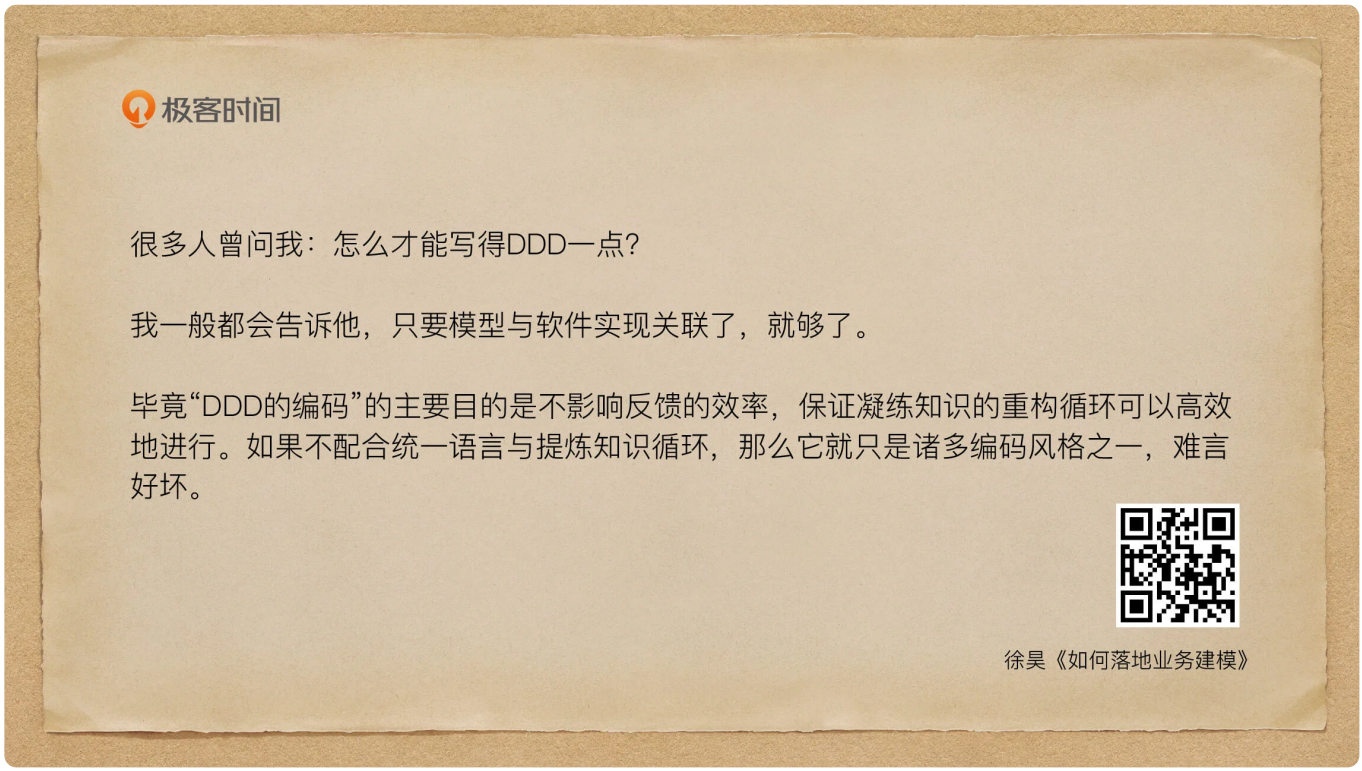
知识消化是一种迭代改进试错法，它并不追求模型的好坏，而是通过**迭代反馈的方式**逐渐提高模型的有效性。这个过程的前提是将模型与软件实现关联在一起。

这种做法在 21 世纪初颇有难度，不过随着工具与框架的成熟，也成为了行业熟知的一种做法。于是，通过迭代反馈凝练知识就变成了实施 DDD 的重点。

不过，在进入这部分之前，我们还要看看如何将统一语言与模型关联起来，这个我们下节课再深入讨论。

思考题

既然领域驱动设计是一种模型驱动的设计方法，为什么不能让业务方直接去使用模型，而要通过统一语言？这是不是有点多余？



欢迎把你的想法和思考分享在留言区，和我一起交流。相信经过你的深度思考，知识会掌握得更牢固。

分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 10 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 开篇词 | 为什么你需要学习业务建模？

下一篇 02 | 统一语言是必要的吗？

精选留言 (28)

写留言

Jxin 置顶
2021-06-24

课后题，个人YY：
1.文中其实没有对“统一语言”下定义。但以我的认知，“统一语言”是什么，确实不好明确定义，它并非一种固定格式的交付物，而是贯穿整个领域驱动设计时的多种元素。是沟通时达成共识的桥梁；是圈定限界上下文时划分聚合的依据；是设计领域模型时定义模型的参考；甚至是命名类名方法名的标准。...
展开

> 作者回复: 问题在后续课程里有解释。至于先有模型还是先有统一语言，这里有区别 就是统一语言提案和统一语言是有区别的，第二课会讲

3 4

Oops!
2021-06-24

不是很理解文中的例子，按例子说的设计方式，模型User要聚合不同业务场景下的不同实体，去订单、浏览记录、评论等，这样一个user模型不会变成巨无霸了吗？还是说，不同领域下都有自己的user模型，每个领域下的user模型聚合的实体都不一样？比如售卖领域，有名为User模型，他聚合了订单这个实体。评论领域，也有个名为User的模型，聚合

了评论，提供了发布评论的接口？如果是这样的话，实际代码实现上，会不会出现大量...

展开

作者回复: 等看第6课 可以解决你的问题

5 9

garlic

2021-06-24

业务操作模型的话，会导致模型和软件割裂，最终变为分析模型。更可怕的是一天一个想法，搞不好搞出百八十个模型都有可能。统一语言起到了屏蔽隔离保护作用，有点像网络分层，具体模型和软件实现由开发人员选取合适的方式实现。

作者回复: nice

9

冯

2021-06-24

我觉得，用统一语言的原因是，模型不好直接用语言描述，一个东西连名字都没有，怎么讨论呢

作者回复: 这是统一语言的一个作用

4

锤他

2021-06-24

思考题： 个人认为根本原因是需要团队协作。首先模型是需要迭代的，迭代通常业务和技术一起讨论，讨论时每个人因为角色/背景/团队等的差异，会对同一个模型产生不同的理解。 通过统一语言，可以把对模型的认识和理解的差异消除。

作者回复: good point

1 4

Smile

2021-06-25

贫血对象模型”（Anemic Model）的实现风格，即：对象仅仅对简单的数据进行封装，而关联关系和业务计算都散落在对象的范围之内。这句话是不是应该在对象之外呀

展开

作者回复: 对。散落之外

2

Vincent

2021-06-25

统一语言我理解是统一理想与现实。业务的模型很多都是从业务角度出发，很多在技术上都很难实现或是不可实现，这个模型需要对现实做一些妥协或是转换，技术上需要对它做一个精简或是明确的定义功能，这样业务与技术达成了一致，后续可以不断迭代这个模型。

作者回复: 实现的方法在4-6里会讲

2

莫太咸

2021-06-25

业务人员直接使用DDL算不算是直接使用模型呢？如果算的话，优点在于，如果模型质量高，DDL简单易用，通过DDL直接使用模型反而能大大提高效率。但缺点是，国内的业务人员有编程思维的人太少；DDL带来的多语言混杂重构调优不易；会有很多重要的领域概念被埋没在DDL的脚本中，导致模型僵化。

展开

作者回复: 算是一种情况，7-9课有一些解法

2

Geek_cbdaf6

2021-06-24

统一语言，我理解是为了更好的做模型淬炼的循环。因为这个过程需要业务方，技术方一起参与，只有统一了语言，在沟通上才会更高效

展开

作者回复: good work

2

酋长的大帝
2021-06-24

个人理解，业务方是站在纯业务的角度提出需求，开发和分析的首要任务是将具体的业务需求抽象成模型，今儿对齐开发。由于开发和分析来抽象出模型，那么语言的选择也是交给开发会更适合。

作者回复: 语言的形成稍微复杂一点

2

邓志国
2021-06-23

业务方使用模型，无法掌握模型的代码，他们只能设计模型的空壳。后续的代码补充和修改就无法和他们同步

作者回复: 两关联一循环就是尝试解决这个问题

2

tongzh
2021-06-25

统一语言是为了减少沟通成本，是为了更容易解释和维护模型。

1

付剑锋
2021-07-19

使用统一语言更容易团队协作和充分讨论和沉淀:
如模型是循环和演进的，团队成员也是变化的，特别是复杂业务要面对多端多业务，可能有多个业务方。如果那模型来使用，个人觉得彼此沟通学习成本太高；而使用统一语言大家可以充分讨论后，再反观现有模型，进行完善扩充和修改。

展开

作者回复: nice

2021-07-08

套用专栏中的说法，在能力评估的过程中，构造一种评估模型，将评估相关的流程和功能转化成模型的行为，从而与业务方同意认知。

业务方和技术方使用统一语言，来描述、讨论需求，从中凝练知识形成领域模型，用领域模型驱动业务开发实现，同时从领域模型中提取出统一语言。...

展开

作者回复: 下一节讲

1

刀刀

2021-07-04

请问徐老师，你觉得现在国内很多软件的代码还是贫血模型主要是什么原因呢？

作者回复: 因为不会

磊

2021-07-04

统一语言的目的是统一认知，是项目参与者之间交流的桥梁。单单只有领域模型是达不到这个效果的。

老狗

2021-07-02

业务逻辑与技术实现之间阻抗，假如业务人员可以直接使用模型，势必对其技术知识提出较高要求，先不说现实与否，分工不存在了

展开

作者回复: 业务人员不会实现代码

奔跑的蜗牛

2021-07-02

统一语言存在的价值是尽量确保，我说的意思，也是你说的意思，大家是在一个意思的基础上提炼的模型。

lzh

2021-06-29

请问老师，聚合根如何理解会容易些，网上找过一些贴子，感觉讲的也很玄乎。是组成业务整体的一组对象中必不可少的对象就叫聚合根吗

展开

作者回复: 第一课有讲

webmin

2021-06-27

同样的字或话语，在不同时代或者不同的地区，因为背景的不一样，可能会具有差别比较大的含义，就连圣经中的上帝都会通过把语言搞得区别开，让人类语言不通无法协作，从而让巴别塔半途而废，由此可见统一语言的必要性。

每个人头脑中对的模型理解，我想就算不是千差万别，也不会是一目了然，再者好多人...

展开

作者回复: 共识是重要的产出物